MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Memo No. 435

January 1978

AMORD

A DEDUCTIVE PROCEDURE SYSTEM

by

Johan de Kleer, Jon Doyle*,
Charles Rich, Guy L. Steele Jr.**, and Gerald Jay Sussman

Abstract:

We have implemented an interpreter for a rule-based system, AMORD, based on a non-chronological control structure and a system of automatically maintained data-dependencies. The purpose of this paper is to serve as a reference manual and as an implementation tutorial. We wish to illustrate:
{1} The discipline of explicit control and dependencies,
{2} How to use AMORD, and
{3} One way to implement the mechanisms provided by AMORD. This paper is organized into sections. The first section is a short "reference manual" describing the major features of AMORD. Next, we present some examples which illustrate the style of expression encouraged by AMORD. This style makes control information explicit in a rule-manipulable form, and depends on an understanding of the use of non-chronological justifications for program beliefs as a means for determining the current set of beliefs. The third section is a brief description of the Truth Maintenance System employed by AMORD for maintaining these justifications and program beliefs. The fourth section presents a complete annotated interpreter for AMORD, written in MacLISP.

* Fannie and John Hertz Foundation Fellow
** NSF Fellow

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by the National Science Foundation under grant MCS77-04828.

Acknowledgements:

We thank Drew McDermott, Richard Stallman and Carl Hewitt for suggestions, ideas and comments used in this paper. Jon Doyle is supported by a Fannie and John Hertz Foundation graduate fellowship. Guy Steele is supported by a National Science Foundation graduate fellowship. This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643.

Contents:

Section 1: The AMORD Reference Manual

AMORD[AMORD] is a system for writing problem solvers. AMORD encourages a style of expression in which the logical relationships of the knowledge and control structure of the problem solver are made explicit. A minimal set of mechanisms is supplied by AMORD so that most of the knowledge that must be formalized and the decisions that must be made in constructing a problem solving program must, to a large degree, be made explicit in AMORD. This makes AMORD is a vehicle for expressing the structure of problem solvers. Once the problem solving structure has been formalized, the task of transferral to programs in programming languages is straightforward. The important aspect of AMORD is the discipline of explicit control it enforces, rather than the specific language or syntax in which the control knowledge is expressed.

The basic mechanism of AMORD is the pattern-directed invocation of a set of rules operating on an indexed data base of assertions. AMORD features a simple syntax for rule invocation patterns, an unconstrained format for assertions, unification semantics for the pattern-matcher, a non-chronological control structure for rule invocations, and the use of a truth maintenance system[TMS] for determining the current set of believed assertions. AMORD is implemented in MacLISP.[MacLISP]

The main components of AMORD are two discrimination networks, one for storing assertions and one for storing rules, the TMS, the matcher, and the queue. The TMS is a system for maintaining the logical grounds for belief in assertions. The matcher is a syntactic unifier which has no distinguished positions or keywords. The queue is a system whereby rules are run on the appropriate assertions. The main loop of the AMORD interpreter is to simply run the body of all rules on all currently believed assertions whose patterns match the rules' patterns. This is done independent of the chronological order in which the assertions and rules are entered into the data bases. When all rules have been run on all matching facts, AMORD halts, awaiting further user input.

There are several special constructs in AMORD for expressing rules and assertions. We will enumerate them here, accompanied by their syntax and description. In these descriptions, expressions of the form "<...>" denote meta-syntactic variables.

ASSERT -- (ASSERT <PATTERN> <JUSTIFICATION>)
This is the method for adding a new assertion (also called a "fact") to the data base. Any variables in the arguments inherit their values from the lexically surrounding text. Variables are denoted by atoms with a colon prefix, as in ":F". Each fact in the data base has an atomic factname. Assertions which are variants of each other denote the same fact in the data base, that is, are mapped to the same factname. The justification is a list, whose interpretation is determined by the first element of the list. If the first element is atomic and has a "proof-type"

function associated with it, that function is applied to the justification and assertion to construct the desired TMS justification. Otherwise, belief in the assertion is justified by belief in all of the facts in the rest of the justification. The addition of a new assertion to the data base causes all rules with patterns matching the assertion to be run.

RULE -- (RULE (<FACTNAME-VARIABLE> <PATTERN>) <BODY>)

This is the method for adding rules to the rule data base. A rule is a procedure to be invoked by all assertions matching <PATTERN>. When a fact whose pattern unifies with the rule pattern is ASSERTed, the set of AMORD and LISP forms specified in the body of the rule are evaluated in the environment specified by adding {1} the variable bindings derived from the unification of the fact pattern and rule pattern to {2} the binding of the fact's factname and the factname variable of the rule pattern and {3} the bindings derived from the lexically surrounding (AMORD, not LISP) text.[Godel] The primary use of the factname variable is for use in specifying justifications in assertions made in the rule body. Rules are run on all matching facts. The order in which they are run is not specified, although the interpreter of Section 4 can be observed to operate in a quasi-depth-first fashion.

ASSUME -- (ASSUME <PATTERN> <JUSTIFICATION>)

This is used to assert speculative hypotheses, that is, to assume a truth "for the sake of argument". Here the <JUSTIFICATION> should specify support for the need for assuming the <PATTERN> assertion. Assumptions are made by justifying belief in the assumed assertion on the basis of a lack of belief in the assumed assertion's negation. Thus, assumptions may be discarded by justifying belief in the negation of the assumed assertion, which invalidates the justification previously supporting belief in the assumed fact. In particular, the dependency-directed backtracking mechanism of the TMS uses the information gained through analysis of the reasons for contradictions to retract conflicting assumptions in this manner.

The following macros can be used to interface expressions manipulated by the AMORD and LISP interpreters.

PDSVAL -- (PDSVAL <FORM>)

This macro allows LISP code to access the AMORD value of <FORM>, that is, the value of all variables prefixed by colons are substituted into the returned form.

PDSLET -- (PDSLET ((<VAR1> <VAL1>) ... (<VARN> <VALN>)) <BODY>)

This macro enables the binding of a number of AMORD variables to values expressed by LISP expressions. Note that the AMORD variables must be prefixed by a colon.

PDSCLOSE -- (PDSCLOSE <BODY>)

This macro allows the evaluation of AMORD forms from within LISP when

the LISP expression being evaluated is not lexically surrounded by an AMORD expression. The forms in the body are evaluated in an empty AMORD environment, that is, an environment in which no AMORD variables are bound.

CONSTANT -- (CONSTANT <OBJECT>)
This LISP predicate determines whether an object contains any references to AMORD variables.

The following are used to initialize and invoke the AMORD interpreter.

INIT -- (INIT)
This function initializes the data bases and various system variables.

RUN -- (RUN)
This function initiates the AMORD read-evaluate loop. Forms read in this loop are closed in the empty environment and then evaluated. Unlike the LISP read-evaluate-print loop, the results of the evaluation of forms in this loop are not printed.

STOP -- (STOP)
This function when read by the AMORD read-evaluate loop causes the loop to halt and return to LISP. AMORD can be invoked again without loss of information by calling RUN, as above.

↑A -- ↑A
This interrupt character (Control-A) performs the same function as STOP above. If typed while AMORD is running, this character causes the loop to halt at the next available point. The queues are left intact, so ↑A(RUN) is a no-op.

The following functions the dependency structures and the data base.

WHY -- (WHY <FACTNAME>)
This prints the current justification for belief in the specified fact.

EXPLAIN -- (EXPLAIN <FACTNAME>)
This prints the complete proof of belief in the specified fact.

PROOFS -- (PROOFS <FACTNAME>)
This prints each of the currently valid justifications for belief in the specified fact.

INSPECT -- (INSPECT '<PATTERN>)
This function prints all of the assertions with patterns matching the given pattern. Each assertion is printed with its factname and, if it is believed, its current justification.

There are also a number of functions internal to the interpreter which are useful in writing specialized functions. The TMS functions and their use are described in Section 3. The most important are the following.

ASSERTION -- (ASSERTION '<PATTERN>)
This returns the factname of the fact with the designated pattern.

FACT-STATEMENT -- (FACT-STATEMENT <FACTNAME>)
This returns the pattern associated with the designated fact.

RETRACT -- (RETRACT <FACTNAME>)
This removes all PREMISE type justifications possessed by the supplied fact.

There are several standard forms of justifications built into AMORD. These are for use in the justification field of ASSERT and ASSUME.

PREMISE -- (PREMISE)
This justification supports belief independent of any other beliefs.

GIVEN -- (GIVEN)
A synonym for PREMISE.

CONDITIONAL-PROOF -- (CONDITIONAL-PROOF <CONSEQUENT> <HYPOTHESES>)
This justification provides support if the current set of justifications for facts provide for belief in the consequent when all the hypotheses are believed. Actually, this justification type has a somewhat more complex capability and syntax which consistently extend the syntax and function just described. The concepts involved in this extension are described in Section 3, and the syntax is described in the annotated implementation in Section 4.

CP -- (CP <CONSEQUENT> <HYPOTHESES>)
A synonym for CONDITIONAL-PROOF.

CONTRADICTION -- (CONTRADICTION <SUPPORT>)
This justification declares the fact justified by this justification to be a contradiction. It supports belief in the justified fact if all the facts mentioned in <SUPPORT> are believed. The declaration of the contradiction will cause backtracking to be invoked whenever the justified fact is believed. All contradictions must be explicitly declared. That is, asserting facts which syntactically are negations of each other does not automatically produce a contradiction.

In addition to the above justification types, the justification types ASSUMPTION, INSTANCE and RULE are used internally by the interpreter in making hypothetical assumptions, in making justifications based on subsumption of one fact by another, and in justifying rules. These justification types should therefore be avoided by the user.

To use AMORD, simply incant at DDT (on MIT-AI):

:AMORD

or

AMORD↑K,

which will load up the current version of AMORD and enter the LISP read-evaluate-print loop. To enter the AMORD read-evaluate loop, evaluate the form (RUN), which will begin interpretation. To escape to LISP, type ↑G, or (STOP) or ↑A as described above.

This concludes the AMORD reference manual.

Section 2: Some AMORD Examples

The structure of AMORD encourages a certain style of rule-writing. In order to compute anything, the control of the computational process must be made explicit.[Explicit Control] The use of explicit control requires careful thought about making the correct justifications for belief in assertions. This section presents a simple deductive system in AMORD to illustrate these points.

The forward version of conjunction introduction can be implemented in AMORD as the following rule:

```
(RULE (:F :A)
      (RULE (:G :B)
            (ASSERT (AND :A :B) (&+ :F :G))))
```

This rule may be paraphrased as follows: the addition of a fact A with factname F into the data base results in the addition of a rule which takes every fact B in the data base and asserts the conjunction of A and B. Thus if FOO is asserted, so will be (AND FOO FOO), (AND FOO (AND FOO FOO)), (AND (AND FOO FOO) FOO), etc. Note that the atom AND is not a distinguished symbol.

Unfortunately, this rule is useless, as it generates piles of useless assertions. To control these deductions, the above rule can be replaced by the following rule which performs consequent reasoning about conjunctive goals.

```
(RULE (:G (SHOW (AND :P :Q)))
      (RULE (:C1 :P)
            (RULE (:C2 :Q)
                  (ASSERT (AND :P :Q) (&+ :C1 :C2)))
            (ASSERT (SHOW :Q) ((BC &+) :G :C1)))
      (ASSERT (SHOW :P) ((BC &+) :G)))
```

In this rule the control statements (those of the form (SHOW ...)) depend on belief in the relevant controlled facts so that the existence of a subgoal for the second conjunct of a conjunctive goal depends on the corresponding solution to the first conjunct. At the same time, no controlled assertions depend on control assertions, since the justification for a conjunction is entirely in terms of the conjuncts, and does not involve the need for deriving the conjunction. This means that the control over the derivation of facts cannot affect the truth of the derived facts. The hierarchy of nested, lexically scoped rules allows the specification of sequencing and restriction information for deriving new assertions. For instance, an alternative method of conjunctive subgoaling can be written as

```
(RULE (:G (SHOW (AND :P :Q)))
      (RULE (:C1 :P)
            (RULE (:C2 :Q)
                  (ASSERT (AND :P :Q) (&+ :C1 :C2))))
      (ASSERT (SHOW :P) ((BC &+) :G))
      (ASSERT (SHOW :Q) ((BC &+) :G)))
```

This rule also only derives correct statements, but is not as tightly controlled as the previous rule. In this case, both subgoals are asserted immediately, although there is no reason to work on the second conjunct unless the first conjunct has been solved. This form of the rule allows more work to be done because possible mutual constraints between the conjuncts due to shared variables are not exploited. That is, in the first consequent rule, solutions to the first conjunct were used to specialize the subgoals for the second conjunct, so that the constraints of the solutions to the first are accounted for in the second subgoal. In the second form of the rule much work might be done on solving each subgoal independently, with the derivation of the conjunction performed by an explicit matching of these derived results. This allows solutions to the second subgoal to be derived which cannot match any solution to the first subgoal.

Other consequent rules for Modus Ponens, Negated Conjunction Introduction, and Double Negation Introduction are similar in spirit to the rule for Conjunction Introduction:

```
(RULE (:G (SHOW :Q))
      (RULE (:I (-> :P :Q))
            (RULE (:F :P)
                  (ASSERT :Q (MP :I :F)))
            (ASSERT (SHOW :P) ((BC MP) :G :I))))

(RULE (:G (SHOW (NOT (AND :P :Q))))
      (RULE (:T (NOT :P))
            (ASSERT (NOT (AND :P :Q)) (-&+ :T)))
      (RULE (:T (NOT :Q))
            (ASSERT (NOT (AND :P :Q)) (-&+ :T)))
      (ASSERT (SHOW (NOT :P)) ((BC -&+) :G))
      (ASSERT (SHOW (NOT :Q)) ((BC -&+) :G)))

(RULE (:G (SHOW (NOT (NOT :P))))
      (RULE (:F :P)
            (ASSERT (NOT (NOT :P)) (--+ :F)))
      (ASSERT (SHOW :P) ((BC --+) :G)))
```

The following two rules implement a consequent oracle for testing the equality of constants. Note the use of PDSVAL in allowing LISP access to the value of AMORD variables.

```
(RULE (:Q (SHOW (= :A :B)))
      (LET ((A (PDSVAL :A))
            (B (PDSVAL :B)))
           (IF (CONSTANT A)
               (IF (CONSTANT B)
                   (IF (EQUAL A B)
                       (ASSERT (= :A :B) (EQUALITY)))))))

(RULE (:Q (SHOW (NOT (= :A :B))))
      (LET ((A (PDSVAL :A))
            (B (PDSVAL :B)))
           (IF (CONSTANT A)
               (IF (CONSTANT B)
                   (IF (EQUAL A B)
                       NIL
                       (ASSERT (NOT (= :A :B)) (EQUALITY)))))))
```

A final example is the use of assumptions to implement a default series of alternative choices. The following expresses the knowledge that traffic signals are either red, yellow or green.

```
(RULE (:T (TYPE :L TRAFFIC-SIGNAL))
      (ASSUME (COLOR :L GREEN) (OPTIMISM :T))
      (RULE (:NG (NOT (COLOR :L GREEN)))
            (ASSUME (COLOR :L YELLOW) (HOPE-YET :T :NG))
            (RULE (:NY (NOT (COLOR :L YELLOW)))
                  (ASSERT (COLOR :L RED) (RATS :T :NG :NY)))))
```

By using this rule, anything declared to be a traffic signal will be assumed to be green in color. If it is discovered (perhaps due to a contradiction) that the color is not green, the color will be assumed to be yellow. If it is further discovered that the color is also not yellow, the color is determined to be red. After creating a number of such traffic signals, their colors can be determined by interrogating AMORD with

```
(INSPECT '(COLOR :X :Y)).
```

Section 3: The Use of the TMS in AMORD

The Truth Maintenance System is an independent program for recording information about program deductions. The TMS uses a method for representing knowledge about beliefs, called a non-monotonic dependency system, to effect any updating of beliefs necessary upon the addition of new information.

The basic operation of the TMS is to attach a justification to a TMS-node. A TMS-node can be linked with any component of program knowledge which is to be connected with other components of program knowledge. In AMORD, each fact and rule has an associated TMS-node. The TMS then decides, on the basis of the justifications attached to nodes, which beliefs in the truth of nodes are supported by the recorded justifications. A node is said to be *in* if there is an associated justification which supports belief in the node. Otherwise, the node is said to be *out*. The TMS informs AMORD whenever the belief status of a node changes, either from *in* to *out*, or *out* to *in*.

There are several types of justifications supported by the TMS. The basic form of a justification is one in which a node is justified if each node in a set of other nodes is *in*. This type of justification represents the typical form of a deduction, or in the special case in which the set of other nodes is empty, a premise. A node may also be justified on the basis of the conditional proof of one node relative to a set of other nodes. In this, belief in the justified node is supported if the consequent node of the conditional proof is *in* when each of the nodes in the set of hypotheses is *in*. The remaining form of justification supports belief in a node if each node in a given set of other nodes is *out*. This non-monotonic justification allows the consistent representation and maintenance of hypothetical assumptions. Using this latter form of justification, a fact can be assumed to be true by justifying it on the basis of its negation being *out*.

Each node which is *in* has a distinguished element of its set of justifications. This distinguished justification is selected to support belief in the node in terms of other nodes having well-founded support, that is, non-circular proofs from ground hypotheses. A number of dependency relations are determined from these justifications, such as the set of nodes depending on a given node, or the nodes upon which a particular node depends.

Truth maintenance processing is required when new justifications cause changes in previously existing beliefs. In such cases, the status of all nodes depending on the nodes with changed beliefs must be redetermined. The critical aspect of this processing is ensuring that all nodes judged to be *in* are associated with well-founded support. Truth maintenance is reminiscent of a generalized and incremental garbage collection. The first step is to mark and collect all facts whose current belief state depends,

via the previously recorded consequence dependencies, on the changed beliefs. The second step is a combination sweep and depth first search over these facts with the purpose of determining belief states based on other facts with well-founded support. By distinguishing facts with well-founded support from those without, all new beliefs determined in this pass are guaranteed to be well-founded. The third step is necessary if the second step does not determine belief states for all the involved facts. This step consists of a relaxation process of assuming some belief states and proceeding, taking care that the assumed beliefs are consistent. This step, at its conclusion, can guarantee that all beliefs have well-founded support. The fourth step is a pass over all changed facts to check for believed facts which are known to represent contradictions. Backtracking is invoked on any such contradictions (which may so invoke further truth maintenance). The final step of truth maintenance is the notification of the external systems of all changes in beliefs determined by the truth maintenance system.

The TMS provides automatic dependency-directed backtracking whenever nodes marked as contradictions are brought *in*. Dependency-directed backtracking employs the recorded dependencies to locate precisely those hypotheses relevant to the failure and uses the conditional proof mechanism to summarize the cause of the contradiction in terms of these hypotheses. Because the reasons for the failure are summarized in a form which is independent of the hypotheses causing the failure, future occurrences of similar failures are avoided.

The TMS functions used in AMORD are as follows:

TMS-MAKE-DEPENDENCY-NODE -- (TMS-MAKE-DEPENDENCY-NODE <EXTERNAL-NAME>)

This function creates a new TMS-node with a given name. In AMORD, the external names are just the atomic factnames used to represent facts and rules. TMS-nodes are currently implemented using uninterned atomic symbols.

TMS-JUSTIFY -- (TMS-JUSTIFY <NODE> <INSUPPORTERS> <OUTSUPPORTERS> <ARGUMENT>)

This function gives a TMS node a new justification, which is valid if each of the nodes of the *in*supporters list is *in*, and each of the nodes of the *out*supporters list is *out*. The argument is an uninterpreted slot used to record the external form of the justification, and is retrievable via the TMS-ANTECEDENT-ARGUMENT function described below.

TMS-CP-JUSTIFY

-- (TMS-CP-JUSTIFY <NODE> <CONSEQUENT> <INHYPOTHESES> <OUTHYPOTHESES> <ARGUMENT>)

This gives a TMS node a new justification which is valid if the consequent node is believed when the *in*hypotheses are *in* and the *out* hypotheses are *out*. As in TMS-JUSTIFY, the argument is an uninterpreted record of the external form of the justification.

TMS-PROCESS-CONTRADICTION
-- (TMS-PROCESS-CONTRADICTION <NAME> <NODE> <TYPE> <CONTRADICTION-FUNCTION>)
This declares a TMS node to represent a contradiction. The name and type are uninterpreted mnemonics provided by the external system to be printed out during backtracking. The contradiction-function, if supplied, should be a LISP function to be called with the contradiction node as its argument when the backtracker can find no backtrackable choicepoints.

TMS-SUPPORT-STATUS -- (TMS-SUPPORT-STATUS <NODE>)
This function returns the support-status, either 'IN or 'OUT, of a node.

TMS-ANTECEDENT-SET -- (TMS-ANTECEDENT-SET <NODE>)
This function returns the list of justifications of the node. In the TMS, each justification is called an antecedent of the node.

TMS-SUPPORTING-ANTECEDENT -- (TMS-SUPPORTING-ANTECEDENT <NODE>)
This function returns the current justification of the node.

TMS-ANTECEDENT-ARGUMENT -- (TMS-ANTECEDENT-ARGUMENT <ANTECEDENT>)
This function returns the external argument associated with the given antecedent.

TMS-ANTECEDENTS -- (TMS-ANTECEDENTS <NODE>)
This function returns the list of nodes determining well-founded support for the given node. This list is extracted from the supporting-antecedent if the node is *in*, and is empty if the node is *out*.

TMS-CONSEQUENCES -- (TMS-CONSEQUENCES <NODE>)
This function returns the list of nodes whose list of antecedent nodes mentions the given node.

TMS-EXTERNAL-NAME -- (TMS-EXTERNAL-NAME <NODE>)
This function returns the user-supplied name of a node.

TMS-IS-IN -- (TMS-IS-IN <NODE>)
This predicate is true iff the node is *in*.

TMS-IS-OUT -- (TMS-IS-OUT <NODE>)
This predicate is true iff the node is *out*.

TMS-RETRACT -- (TMS-RETRACT <NODE>)
This function will remove all premise-type justifications from the set of justifications of the node.

TMS-PREMISES -- (TMS-PREMISES <NODE>)
This function returns a list of the premises among the well-founded support of the node.

TMS-ASSUMPTIONS -- (TMS-ASSUMPTIONS <NODE>)
This function returns a list of the assumptions among the well-founded support of the node.

The TMS also generates new "facts" internally during backtracking. These will therefore occur in explanations and antecedents of the nodes requested and justified by the external systems. The internal facts generated by the TMS are atoms with certain properties. The following functions are provided to manipulate these internal facts.

TMS-FACTP -- (TMS-FACTP <THING>)
This predicate is true iff the thing is an internal TMS fact.

TMS-FACT-NODE -- (TMS-FACT-NODE <FACT>)
This function returns the TMS node associated with an internal fact.

TMS-FACT-STATEMENT -- (TMS-FACT-STATEMENT <FACT>)
This function returns the symbolic statement of the meaning of an internal fact. This statement refers to the external names of the other facts, such as contradictions and assumptions, which were involved in the making of the fact.

The following two functions are supplied for debugging purposes.

TMS-INIT -- (TMS-INIT)
This function clears the state of the TMS by resetting all internal variables and clearing all properties and internings of TMS nodes.

TMS-INTERN -- (TMS-INTERN)
This function interns all TMS nodes currently in existence, and causes the interning of all nodes generated in the future. Initially, the atomic symbols representing TMS nodes are not interned.

Examples of the use of the TMS facilities can be found in the following section, in which the functions implementing the various AMORD proof-types are defined.

Section 4: An Annotated Interpreter

Here we present an actual AMORD interpreter. The interpreter divides into the following sections, which will be presented in this order.

AMORD form definitions
    ASSERT and associated functions
    RULE and associated functions
Proof-type definitions
The RUN interpreter (the main loop)
The TMS interface
The Unification Matcher
The Discrimination-Net Data Base

Before presenting the interpreter itself, we describe some aspects of the implementation.

The main loop of the interpreter is in the function RUN, which examines the various queues (described below). RUN makes sure that all rules are run on all facts whose patterns match the rule patterns. As an efficiency measure, a rule is run on a fact only if both the rule and fact are believed (*in*). After the possibilities for running rules on facts are exhausted, RUN checks for programs (called "runlast" functions) which have been specified for running at queue's end and runs each of these programs. If these programs make new assertions or rules, the above loop is resumed. Finally, after finishing all of the above steps, RUN prints out a prompt string and waits for new input from the user.

Each rule and fact is represented by an atomic symbol. The information used by AMORD is stored in a data structure kept as the value of the atomic symbol. In these data structures are the TMS-nodes of the rules and facts and the "stimulate-lists", which store matching facts and rules (respectively) until they are queued up to be run.

In addition, rules and facts have other attached items. Facts have their statement, and rules have their full trigger pattern (the list of the factname variable and the trigger pattern proper). Rules are distinguished from facts by their possession of an extra data structure containing the uninstantiated rule body and the environment of AMORD variable bindings derived from the lexically surrounding text.

The control of running rules on facts is mediated by an amorphous mechanism called the queue. This mechanism has several components:

{1} The trigger queue, *TQ*. This is a queue of rule-fact pairs representing possible triggerings. This queue is maintained, in the global variable *TQ*, as a CONS cell, the CAR of which points to the front of the list of trigger pairs, and the CDR of which points to the last cell of this list. This is done so that new pairs may be quickly added to the end of the list of trigger pairs.

{2} The stimulate lists. Each rule and fact has a list, of facts and rules respectively called its "stimulate-list". These facts and rules in these lists are initially the items retrieved from the data base as possibly matching the newly created rule or fact. The function STIMULATE, called by the TMS when rules and facts come *in*, takes the stimulate-list of the newly *in*ned item, turns it into a list of pairs and adds these pairs to the trigger queue.

The queue mechanism operates as follows. When pairs come to the top of the trigger queue, both the rule and the fact of the pair are checked to see if they are *in*. If both are *in*, their unification is attempted. (The matching done by the data base fetch routines only provides candidates for the true unification match.) If they do not unify, the pair is discarded from the queueing system: if they do, the rule body is evaluated in the derived environment. Alternatively, if a pair is encountered on the trigger queue with the rule (or fact) *out*, the fact (or rule) is placed on the STIMULATE-LIST of the *out* rule (or fact). In this way {1} pairs are not run until they become relevant, and {2} pairs are run at most once, for subsequent *in*nings of the rules or facts involved will keep adding the pair to the trigger queue until the pair makes it to the top with both items *in*, at which time the pair will run and leave the queue system.

In addition to the above trigger queue mechanism, two other structures are part of the main RUN loop.

{1} The closure queue, *Q*. This is queue of arbitrary LISP forms to be evaluated. The global variable *Q* contains this queue, in the form of a CONS whose CAR is the first cell of the list forming the queue, and whose CDR is the last cell of this list. As in the trigger queue, this is done so that new queue items can be added directly at the end of the queue, rather than requiring a traversal through the entire queue for each new addition. This queue is provided so that the user may post programs to be executed. This is sometimes (although rarely) necessary, as the TMS makes the restriction that the TMS cannot be invoked while a previous invocation is still signalling changes in the statuses of facts.

{2} The runlast list, *RUNLAST*. This is a user maintained list, initially empty, of LISP forms to be evaluated each time both *TQ* and *Q* run out. At such time, each form in this list is evaluated. These forms can either add new justifications to facts, add other programs to *Q* to be run, or, by means of PDSCLOSE, evaluate further AMORD forms to cause resumption of the main loop of trigger queue interpretation.

The structure of justifications is as follows. Justifications must be lists. If the first element of the list is either non-atomic, or lacks a 'PROOF-TYPE property if atomic, the justification is interpreted as a simple deductive justification in which the justified item will be *in* if all the facts mentioned in the rest of the justification are *in*. If the first element of the justification is an atom with a 'PROOF-TYPE property, the the value of that property must be a LISP function. This function is called with the justification and justified item as arguments. This function then has the responsibility for making the necessary TMS

justifications, and may perform other operations if desired. Proof-type functions which must evaluate AMORD forms should use the PDSCLOSE macro described in Section 1.

The interpreter uses several global variables as follows:

Q - The queue containing LISP forms to evaluate.
TQ - The trigger queue containing rule-fact pairs to close and run.
ENTRY - Contains the last *Q* form evaluated by RUN.
RUNLAST - A list of LISP forms to be successively evaluated each time the queue runs out. This list is initially NIL.
STOPFLAG - If non-NIL, causes the RUN loop to halt after running the current entry.
SUBSTITUTION - This variable is bound by TRY-RULE to the current AMORD environment to be used in evaluating rule bodies.
T-LIST - This variable is bound by TRY-RULE to a list of the triggering assertion and executing rule for use in justifying subrules.
WALLP - If non-NIL, causes new justifications of assertions to be displayed. The default is T.
RULE-WALLP - If non-NIL and if *WALLP* is also non-NIL, causes new justifications of rules to be displayed. The default is NIL.
DN - Contains the discrimination net.
GENSYM-COUNTER - The counter used in generating rule and fact names, numbers for standardizing expressions apart, and line numbers.

Here begins the code of the interpreter proper. Several macros are used in this code, including the substituting-quote ", which returns the next form, quoted but with the values of subforms preceded by , substituted as elements of list structure, and with the values of subforms preceded by @ spliced in as list segments. The macros DEFMAC, IF, and LET have the obvious meanings, and are defined both during compilation and in the AMORD runtime environment.

The first items are declarations for the MacLISP compiler.

```
(DECLARE (*EXPR TMS-CLOBBER-SIGNAL-RECALLING-FUNCTION TMS-INIT
                TMS-MAKE-DEPENDENCY-NODE TMS-NODE TMS-NODES
                TMS-JUSTIFY TMS-CP-JUSTIFY TMS-PROCESS-CONTRADICTION
                TMS-RETRACT TMS-ASSUMPTIONS TMS-PREMISES TMS-ALL-CONSEQUENCES
                TMS-ALL-ANTECEDENTS TMS-ARE-OUT TMS-ARE-IN TMS-IS-OUT TMS-IS-IN
                TMS-CONSEQUENCES TMS-EXTERNAL-NAME TMS-ANTECEDENTS TMS-ANTECEDENT-SET
                TMS-SUPPORTING-ANTECEDENT TMS-ANTECEDENT-ARGUMENT TMS-SUPPORT-STATUS
                TMS-FACT-NODE TMS-FACT-STATEMENT TMS-FACTP TIMESTAMP)
         (*FEXPR GCTWA)
         (SPECIAL *WALLP* *RULE-WALLP* *STOPFLAG* *TQ* *Q* *ENTRY* *RUNLAST*
                  *GENSYM-COUNTER* *SUBSTITUTION* *T-LIST*))
```

The following macros define the data structures representing rules and assertions. None are defined following compilation. Functions are provided instead.

```
(DECLARE (MACROS NIL))                    ;TURN OFF MACRO RETENTION.

(DEFMAC GET-FACT-STATEMENT (FACT) "(CAAR (SYMEVAL ,FACT)))

(DEFUN FACT-STATEMENT (F)
       (IF (TMS-FACTP F) (TMS-FACT-STATEMENT F) (GET-FACT-STATEMENT F)))

(DEFMAC GET-RULE-PATTERN (RULE) "(CAAR (SYMEVAL ,RULE)))

(DEFMAC RULEP (ITEM) "(CDDR (SYMEVAL ,ITEM)))          ;CHECKS FOR RULE PARTS

(DEFMAC GET-TMS-NODE (ITEM) "(CDAR (SYMEVAL ,ITEM)))

(DEFMAC GET-STIMULATE-LIST (ITEM) "(CADR (SYMEVAL ,ITEM)))

(DEFMAC SET-STIMULATE-LIST (ITEM STIM-LIST)
        "(RPLACA (CDR (SYMEVAL ,ITEM)) ,STIM-LIST))

(DEFMAC GET-RULE-FUNCTION (RULE) "(CADDR (SYMEVAL ,RULE)))

(DEFMAC GET-RULE-SPECIALIZATION (RULE) "(CDDDR (SYMEVAL ,RULE)))

(DEFMAC MAKE-ASSERTION-STRUCTURE (EXP TMS-N STIM-LIST)
        "(CONS (CONS ,EXP ,TMS-N) (CONS ,STIM-LIST NIL)))

(DEFMAC MAKE-RULE-STRUCTURE (PAT TMS-N STIM-LIST RULE-FUN SPEC)
        "(CONS (CONS ,PAT ,TMS-N) (CONS ,STIM-LIST (CONS ,RULE-FUN ,SPEC))))

(DECLARE (MACROS T))                      ;TURN ON MACRO RETENTION.
```

AMORD FORM DEFINITIONS

All true AMORD forms like ASSERT and RULE must be evaluated in a LISP environment in which the variables *SUBSTITUTION* and *T-LIST* are bound. To achieve this, while making these variables invisible to the user, macros are used which append the appropriate variable references to the calls to the AMORD primitives.

Here is ASSERT, which takes an expression and a justification, instantiates them with the current environment bindings, inserts the expression into the data base, and then installs the justification as one of the expression's justifications. The call to SUBSUME-CHECK serves to add new justifications to the new fact or to other facts based on subsumptions in their patterns.

```
(DEFMAC ASSERT (EXPRESSION JUSTIFICATION)
       "(ASSERT-2 ',EXPRESSION ',JUSTIFICATION *SUBSTITUTION*))

(DEFUN ASSERT-2 (EXPRESSION JUSTIFICATION ALIST)
      (LET ((A (ASSERTION (INSTANCE EXPRESSION ALIST))))
           (INSTALL-JUST (INSTANCE JUSTIFICATION ALIST) A)
           (SUBSUME-CHECK A)))
```

The operation of ASSUME is somewhat more complicated than that of ASSERT, as two facts are created in addition to the specified fact, as well as one additional justification.

```
(DEFMAC ASSUME (EXPRESSION JUSTIFICATION)
       "(ASSUME-2 ',EXPRESSION ',JUSTIFICATION *SUBSTITUTION*))

(DEFUN ASSUME-2 (EXPRESSION JUSTIFICATION ALIST)
       (LET ((EXPRESSION (INSTANCE EXPRESSION ALIST)))
            (LET ((A (ASSERTION EXPRESSION))
                  (AF (ASSERTION "(ASSUMED ,EXPRESSION)))
                  (N (ASSERTION
                       (IF (EQ (CAR EXPRESSION) 'NOT)
                           (CADR EXPRESSION)
                           "(NOT ,EXPRESSION)))))
              (INSTALL-JUST (INSTANCE JUSTIFICATION ALIST) AF)
              (INSTALL-JUST "(ASSUMPTION ,AF ,N) A)
              (SUBSUME-CHECK A)
              (SUBSUME-CHECK AF)
              (SUBSUME-CHECK N))))
```

ASSERTION is the function for creating new assertions. The data base is checked to see if it contains a fact with a variant of the supplied pattern. If so, that fact is returned, and otherwise a new fact is generated and inserted into the data base in the appropriate bucket.

```
(DEFUN ASSERTION (EXPRESSION)
       (LET ((B (BUCKET EXPRESSION NIL 'ASSERTION)))
            (DO ((L (STUFF B) (CDR L))
                 (C))
                ((NULL L)
                 (LET ((NAME (GENS 'F)))
                      (SET NAME
                           (MAKE-ASSERTION-STRUCTURE
                            EXPRESSION
                            (TMS-MAKE-DEPENDENCY-NODE NAME)
                            (FETCH EXPRESSION NIL 'RULE)))
                      (INSERT-IN-BUCKET NAME B)
                      NAME))
                (SETQ C (COMPARE EXPRESSION (GET-FACT-STATEMENT (CAR L))))
                (AND C (EQ (CAR C) 'VARIANT) (RETURN (CAR L))))))
```

SUBSUME-CHECK performs the function of checking the data base for facts whose patterns either subsume or are subsumed by the pattern of the supplied fact. If any subsumptions are detected, new justifications are added to support belief in the subsumed fact if the subsuming fact is believed.

```
(DEFUN SUBSUME-CHECK (NAME)
       (LET ((EXP (GET-FACT-STATEMENT NAME)))
            (DO ((CANDIDATES (FETCH EXP NIL 'ASSERTION) (CDR CANDIDATES))
                 (C))
                ((NULL CANDIDATES))
                (COND ((EQ (CAR CANDIDATES) NAME))
                      ((NULL (SETQ C (COMPARE EXP (GET-FACT-STATEMENT (CAR CANDIDATES)))))
                      ((EQ (CAR C) 'SUBSUMES)
                       (INSTALL-JUST (LIST 'INSTANCE NAME) (CAR CANDIDATES)))
                      ((EQ (CAR C) 'SUBSUMED)
                       (INSTALL-JUST (LIST 'INSTANCE (CAR CANDIDATES)) NAME))
                      (T (BREAK |SUBSUME-CHECK|)))))))
```

The next function is not used in the interpreter, but provides a useful service in writing AMORD rules and proof types. PRESENT takes as its argument a full rule pattern of the form (<factname> <pattern>). It returns a list of substitutions corresponding to all matching (subsumed by the pattern) assertions existing in the data base.

```
(DEFUN PRESENT (PATTERN)
       (DO ((CANDIDATES (FETCH (CADR PATTERN) NIL 'ASSERTION) (CDR CANDIDATES))
            (ANS NIL)
            (C))
           ((NULL CANDIDATES) ANS)
           (AND (SETQ C (COMPARE (CADR PATTERN) (GET-FACT-STATEMENT (CAR CANDIDATES))))
                (MEMQ (CAR C) '(SUBSUMES VARIANT))
                (SETQ ANS (CONS (CONS (CONS (CAR PATTERN) (CAR CANDIDATES)) (CADR C))
                                ANS)))))
```

INSPECT applies PRESENT to a useful task. It prints all assertions matching the supplied pattern, in order of ascending factname.

```
(DEFUN INSPECT (PATTERN)
       (SETQ PATTERN "((/: *FACTNAME* . 0) ,PATTERN))
       (MAPC '(LAMBDA (SUB)
                      (LET ((I (INSTANCE PATTERN SUB)))
                           (COND ((IS-IN (CAR I))
                                  (PRINT I)
                                  (PRIN1 (ARGUMENT (CAR I))))
                                 (T (PRINT I)
                                    (PRINC '|(OUT)|)))))
             (SORT (PRESENT PATTERN) 'INSPECT-SORT))
       'DONE)

(DEFUN INSPECT-SORT (X Y)
       (FACT-NAME-ALPHAGREATERP (CDAR X) (CDAR Y)))
```

RULE-PRESENT is like PRESENT but for rules.

```
(DEFUN RULE-PRESENT (PATTERN)
       (DO ((CANDIDATES (FETCH PATTERN NIL 'RULE) (CDR CANDIDATES))
            (ANS NIL)
            (C))
           ((NULL CANDIDATES) ANS)
           (AND (SETQ C (COMPARE PATTERN (CADR (GET-RULE-PATTERN (CAR CANDIDATES)))))
                (MEMQ (CAR C) '(SUBSUMES VARIANT))
                (SETQ ANS (CONS (CONS (CAR CANDIDATES) (CADR C))
                                ANS)))))
```

INSPECT-RULES is like INSPECT but for rules. This pretty-prints the complete rule definitions, so prepare for a *lot* of output.

```
(DEFUN INSPECT-RULES (PATTERN)
       (LET ((L (SORT (RULE-PRESENT PATTERN) 'INSPECT-RULES-SORT)))
            (MAPC '(LAMBDA (SUB)
                           (LET ((I (LIST (CAR SUB)
                                          (INSTANCE (LIST 'RULE
                                                          (GET-RULE-PATTERN (CAR SUB))
                                                          (GET-RULE-FUNCTION (CAR SUB)))
                                                    (CDR SUB)))))
                                (COND ((IS-IN (CAR I))
                                       (SPRINTER I)
                                       (PRINT (ARGUMENT (CAR I)))
                                       (TERPRI)
                                       (TERPRI))
                                      (T (SPRINTER I)
                                         (PRINT '(OUT))
                                         (TERPRI)
                                         (TERPRI)))))
                  L))
       'DONE)

(DEFUN INSPECT-RULES-SORT (X Y)
       (FACT-NAME-ALPHAGREATERP (CAR X) (CAR Y)))
```

Rules have justifications just like facts, but unlike facts, rules are used in no justifications (other than in justifying their subrules). Rules are really operational entities, which should be allowed to function only if the facts leading to their creation (via other rules forming its lexical environment) are believed. This is the purpose of the *T-LIST* mechanism seen below in the functions for defining new rules.

```
(DEFMAC RULE (PATTERN . BODY)
        "(RULE-2 ',PATTERN ',BODY *SUBSTITUTION* *T-LIST*))

(DEFUN RULE-2 (PATTERN RULE-FUNCTION ALIST T-LIST)
       (LET ((B (BUCKET (CADR PATTERN) ALIST 'RULE))
             (RNAME (GENS 'R)))
            (SET RNAME
                 (MAKE-RULE-STRUCTURE
                  PATTERN
                  (TMS-MAKE-DEPENDENCY-NODE RNAME)
                  (FETCH (CADR PATTERN) ALIST 'ASSERTION)
                  RULE-FUNCTION
                  ALIST))
            (INSERT-IN-BUCKET RNAME B)
            (INSTALL-JUST "(RULE . ,T-LIST) RNAME)))
```

TRY-RULE takes a possible triggering pair, consisting of a rule and a fact. The pattern of the fact is compared with the pattern of the rule. If these two patterns unify, then the body of the rule is evaluated in the environment produced by adding the bindings derived from the unification to the environment in which the rule is run.

```
(DEFUN TRY-RULE (RNAME ANAME)
       (LET ((S (UNIFY (CADR (GET-RULE-PATTERN RNAME))
                       (GET-FACT-STATEMENT ANAME)
                       (GET-RULE-SPECIALIZATION RNAME))))
    (IF S
        (LET ((*SUBSTITUTION*
               "((,(CAR (GET-RULE-PATTERN RNAME)) . ,ANAME) . ,(CAR S)))
              (*T-LIST*
               "(,ANAME ,RNAME)))
             (MAPC 'EVAL (GET-RULE-FUNCTION RNAME))))))
```

PROOF-TYPES AND JUSTIFICATIONS

INSTALL-JUST takes a justification and a fact (or rule). If the justification has an associated proof-type, the proof-type function is called with the justification and fact as arguments. Otherwise, SUPPORT is called to add the justification to the set of justifications of the fact. If the new justification causes the fact to be newly believed, the fact and its justification may be displayed.

```
(DEFUN INSTALL-JUST (JUSTIFICATION FACT)
  (LET ((OLDSTATUS (SUPPORT-STATUS FACT)))
       (IF (SYMBOLP (CAR JUSTIFICATION))
           (LET ((G (GET (CAR JUSTIFICATION) 'PROOF-TYPE)))
                (IF G (FUNCALL G JUSTIFICATION FACT) (SUPPORT JUSTIFICATION FACT)))
           (SUPPORT JUSTIFICATION FACT))
       (AND *WALLP*
            (COND ((RULEP FACT)
                   (COND ((AND *RULE-WALLP*
                               (EQ OLDSTATUS 'OUT)
                               (EQ (SUPPORT-STATUS FACT) 'IN))
                          (PRINT 'DEFINING)
                          (PRIN1 FACT)
                          (PRINC '| |)
                          (SPRINTER (INSTANCE (LIST 'RULE
                                                    (GET-RULE-PATTERN FACT)
                                                    (GET-RULE-FUNCTION FACT))
                                              (GET-RULE-SPECIALIZATION FACT)))
                          (PRINC '| |)
                          (PRIN1 JUSTIFICATION)
                          (TERPRI)
                          (TERPRI))))
                  ((AND (EQ OLDSTATUS 'OUT)
                        (EQ (SUPPORT-STATUS FACT) 'IN))
                   (PRINT 'ASSERTING)
                   (PRIN1 FACT)
                   (PRINC '| |)
                   (PRIN1 (GET-FACT-STATEMENT FACT))
                   (PRINC '| |)
                   (PRIN1 JUSTIFICATION))))))

(SETQ *WALLP* T)
(SETQ *RULE-WALLP* NIL)
```

SUPPORT performs the standard task of justification, which interprets all elements of the supplied justification (except the first, which is mnemonic) to be factnames which collectively justify belief in the supplied fact.

```
(DEFUN SUPPORT (JUSTIFICATION FACT)
       (TMS-JUSTIFY (TMS-NODE FACT)
                    (TMS-NODES (CDR JUSTIFICATION))
                    NIL
                    JUSTIFICATION))
```

PREMISE justifies the fact with a eternally valid justification.

```
(DEFUN PREMISE (JUSTIFICATION FACT)
        (TMS-JUSTIFY (TMS-NODE FACT) NIL NIL JUSTIFICATION)))

(PUTPROP 'PREMISE 'PREMISE 'PROOF-TYPE)
(PUTPROP 'GIVEN 'PREMISE 'PROOF-TYPE)
```

CONDITIONAL-PROOF interprets the second element of the justification as the consequent of the conditional proof, the third element as the list of *in* hypotheses of the conditional proof, and the fourth element as the list of *out* hypotheses of the conditional proof.

```
(DEFUN CONDITIONAL-PROOF (JUSTIFICATION FACT)
       (TMS-CP-JUSTIFY (TMS-NODE FACT)
                       (TMS-NODE (CADR JUSTIFICATION))
                       (TMS-NODES (CADDR JUSTIFICATION))
                       (TMS-NODES (CADDDR JUSTIFICATION))
                       JUSTIFICATION))

(PUTPROP 'CP 'CONDITIONAL-PROOF 'PROOF-TYPE)
(PUTPROP 'CONDITIONAL-PROOF 'CONDITIONAL-PROOF 'PROOF-TYPE)
```

ASSUMPTION interprets the second element of the justification as a factname designating the reason for making the assumption, and the third element as a factname designating a negation of the belief to be assumed. Thus the supplied fact will be believed whenever the reason fact is *in*, and the negation fact is *out*.

```
(DEFUN ASSUMPTION (JUSTIFICATION FACT)
       (TMS-JUSTIFY (TMS-NODE FACT)
                    (LIST (TMS-NODE (CADR JUSTIFICATION)))
                    (LIST (TMS-NODE (CADDR JUSTIFICATION)))
                    JUSTIFICATION))

(PUTPROP 'ASSUMPTION 'ASSUMPTION 'PROOF-TYPE)
```

CONTRADICTION first supports belief in the supplied fact and then declares to the TMS that the fact is a contradiction.

```
(DEFUN CONTRADICTION (JUSTIFICATION FACT)
       (SUPPORT JUSTIFICATION FACT)
       (TMS-PROCESS-CONTRADICTION FACT (TMS-NODE FACT) (GET-FACT-STATEMENT FACT) NIL))

(PUTPROP 'CONTRADICTION 'CONTRADICTION 'PROOF-TYPE)
```

THE RUN INTERPRETER

The following three macros hide references to the variables *SUBSTITUTION* and *T-LIST*, allowing LISP and AMORD code to be mixed.

```
(DEFMAC PDSVAL (ID) "(INSTANCE ',ID *SUBSTITUTION*))

(DEFMAC PDSLET (VARS . BODY)
       "(LET ((*SUBSTITUTION*
              ,(DO ((A '*SUBSTITUTION*
                       "(CONS (CONS ',(CAAR VL) ,(CADAR VL)) ,A))
                    (VL VARS (CDR VL)))
                   ((NULL VL) A))))
             @BODY))

(DEFMAC PDSCLOSE BODY "(LET ((*SUBSTITUTION* NIL) (*T-LIST* NIL)) @BODY))
```

RUN has four loops in one. First the trigger queue is tried, then the main queue, then the runlast functions, and finally the reader is invoked. The loop is halted on any iteration if *STOPFLAG* is non-NIL.

```
(DEFUN RUN ()
       (PROG (R F)
             (SETQ *STOPFLAG* NIL)
        LOOP (COND (*STOPFLAG* (RETURN 'STOPPED))
                   ((CAR *TQ*)
                    (SETQ R (CAAAR *TQ*))
                    (SETQ F (CDAAR *TQ*))
                    (RPLACA *TQ* (CDAR *TQ*))
                    (IF (IS-IN F)
                        (IF (IS-IN R)
                            (TRY-RULE R F)
                            (SET-STIMULATE-LIST R (CONS F (GET-STIMULATE-LIST R))))
                        (SET-STIMULATE-LIST F (CONS R (GET-STIMULATE-LIST F))))
                    (GO LOOP))
                   ((CAR *Q*)
                    (SETQ *ENTRY* (CAAR *Q*))
                    (RPLACA *Q* (CDAR *Q*))
                    (EVAL *ENTRY*)
                    (GO LOOP)))
              (DO ((RL *RUNLAST* (CDR RL)))
                  ((NULL RL))
                  (EVAL (CAR RL)))
              (AND (OR (CAR *TQ*) (CAR *Q*)) (GO LOOP))
              (SETQ *GENSYM-COUNTER* (+ *GENSYM-COUNTER* 1))
              (PRINT *GENSYM-COUNTER*)
              (PRINC '|>> |)
              (ENQUEUE (LIST "(PDSCLOSE ,(READ))))
              (GO LOOP)))
```

The following implement the RUN loop controllers.

```
(DEFUN AMORD-RUN-INTERRUPT (X Y) (SETQ *STOPFLAG* T) 'RUN-INTERRUPTED)

(SSTATUS TTYINT '/↑A 'AMORD-RUN-INTERRUPT)

(DEFUN STOP () (SETQ *STOPFLAG* T))
```

ENQUEUE augments *Q* with a list of new forms.

```
(DEFUN ENQUEUE (ACTIONS)
       (IF ACTIONS
           (LET ((L (LAST ACTIONS)))
                (COND ((CAR *Q*)
                       (RPLACD (CDR *Q*) ACTIONS)
                       (RPLACD *Q* L))
                      (T (RPLACA *Q* ACTIONS)
                         (RPLACD *Q* L))))))
```

STIMULATE is the function called by the TMS on any fact or rule which changes status from *out* to *in*. When such a status change takes place, all items on the stimulate list are used to add new pairs to the trigger queue. DESTIMULATE is the complementary function called when assertions or rules go from *in* to *out*. It is ignored by AMORD.

```
(DEFUN STIMULATE (NAME)
       (LET ((ACTIONS (IF (RULEP NAME)
                          (MAPCAR '(LAMBDA (F) (CONS NAME F)) (GET-STIMULATE-LIST NAME))
                          (MAPCAR '(LAMBDA (R) (CONS R NAME)) (GET-STIMULATE-LIST NAME)))))
            (SET-STIMULATE-LIST NAME NIL)
            (IF ACTIONS
                (LET ((L (LAST ACTIONS)))
                     (COND ((CAR *TQ*)
                            (RPLACD (CDR *TQ*) ACTIONS)
                            (RPLACD *TQ* L))
                           (T (RPLACA *TQ* ACTIONS)
                              (RPLACD *TQ* L)))))))

(DEFUN DESTIMULATE (NAME) NIL)
```

INIT performs several functions. It initializes the discrimination net, the TMS, and the global variables of the AMORD system. It also attempts (by a somewhat less than elegant method) to rid the system of all assertions and rules previously created.

```
(DEFUN INIT ()
       (DBINIT)
       (TMS-INIT)
       (SETQ *Q* (CONS NIL NIL)) ;CAR IS FIRST CELL OF QUEUE, CDR IS LAST CELL
       (SETQ *TQ* (CONS NIL NIL))
(SETQ *RUNLAST* NIL)
(SETQ *ENTRY* NIL)
(SETQ *STOPFLAG* NIL)
(COND ((AND (BOUNDP *GENSYM-COUNTER*)
            (NUMBERP *GENSYM-COUNTER*)))
      (T (SETQ *GENSYM-COUNTER* 0)))
((LAMBDA (BASE *NOPOINT)
         (DECLARE (SPECIAL BASE *NOPOINT))
         (DO ((I 1 (1+ I))
              (A))
             ((> I *GENSYM-COUNTER*))
             (SETQ A (READLIST (CONS 'F (CONS '- (EXPLODE I)))))
             (MAKUNBOUND A)
             (SETPLIST A NIL)
             (REMOB A)
             (SETQ A (READLIST (CONS 'R (CONS '- (EXPLODE I)))))
             (MAKUNBOUND A)
             (SETPLIST A NIL)
             (REMOB A)))
 8. T)
(GCTWA)
(SETQ *GENSYM-COUNTER* 0)
'INITIALIZED)
```

Variables are represented by semi-lists of three elements, in the form (/: <var> . <number>) The first element is the atom ":", the second is the variable name, and the third is a number used to standardize the variable name apart. The following functions should be used to test for them.

```
(DEFUN VARIABLE (X) (EQ (CAR X) '/:))
```

CONSTANT tests whether an S-expression contains any variables.

```
(DEFUN CONSTANT (X)
       (COND ((ATOM X) (NOT (EQ X '/:)))
             ((CONSTANT (CAR X)) (CONSTANT (CDR X)))))
```

GENS generates a new atomic symbol with a supplied prefix and a suffix of the form "-nnn".

```
(DEFUN GENS (E)
       (READLIST (NCONC (EXPLODE E)
                        (LIST '-)
                        ((LAMBDA (BASE *NOPOINT)        ;AVOID SCREWS DUE TO BASE CHANGES
                                 (DECLARE (SPECIAL BASE *NOPOINT))
                                 (EXPLODE (SETQ *GENSYM-COUNTER*
                                                (+ *GENSYM-COUNTER* 1))))
                         8. T))))
```

The variable designator ":" is a read macro which generates the standard variable-structure described above. Because items read in see a constant value for *GENSYM-COUNTER*, variable references in an expression (such as two occurrences of ":x") appear as similar structures (such as "(/: x . 127)").

```
(DEFUN COLON-READ () (CONS '/: (CONS (READ) *GENSYM-COUNTER*)))

(SETSYNTAX '/: 'MACRO 'COLON-READ)
```

THE TMS INTERFACE

WHY presents the immediate justification for the current belief in a fact. Note that if the fact is not believed, the list of failing justifications is printed. PROOFS prints all of the justifications possesed by an assertion. EXPLAIN collects up all facts among the support of the supplied fact, sorts them by the suffix of their factname, and prints them one per line along with their current justifications.

```
(DEFUN WHY (NAME)
       (PRINT NAME)
       (PRIN1 (FACT-STATEMENT NAME))
       (PRINC '| |)
       (IF (IS-IN NAME)
           (PRIN1 (ARGUMENT NAME))
           (PRIN1 (CONS 'OUT
                        (MAPCAR 'ARGUMENT (ANTECEDENT-SET NAME)))))
       'QED)

(DEFUN PROOFS (FACT)
       (TERPRI) (PRINC '|PROOFS OF |) (PRIN1 FACT) (PRINC '| = |) (PRIN1 (FACT-STATEMENT FACT))
       (PRINC '| (|) (PRIN1 (SUPPORT-STATUS FACT)) (PRINC '|) |)
       (MAPC '(LAMBDA (A) (PRINT (TMS-ANTECEDENT-ARGUMENT A)))
             (TMS-ANTECEDENT-SET (TMS-NODE FACT)))
       'QED)

(DEFUN EXPLAIN (FACT)
       (TERPRI) (PRINC '|PROOF OF |) (PRIN1 FACT) (PRINC '| = |) (PRIN1 (FACT-STATEMENT FACT))
       (PRINC '| (|) (PRIN1 (SUPPORT-STATUS FACT)) (PRINC '|) |) (PRIN1 (ARGUMENT FACT))
       (PFL (FOUNDATIONS FACT))
       'QED)
```

The following functions do the dirty work for functions like EXPLAIN.

```
(DEFUN PFL (FL)
       (MAPC '(LAMBDA (F)
                (PRINT F)
                (PRINC '| = |)
                (PRIN1 (FACT-STATEMENT F))
                (PRINC '| (|) (PRIN1 (SUPPORT-STATUS F)) (PRINC '|) |)
                (PRIN1 (ARGUMENT F)))
             (SORT (APPEND FL NIL) 'FACT-NAME-ALPHAGREATERP)))

(DEFUN FACT-NAME-ALPHAGREATERP (F G)
       (GREATERP (GENS-NUMBER-EXTRACT F) (GENS-NUMBER-EXTRACT G)))

(DEFUN GENS-NUMBER-EXTRACT (X)
       (DO ((E (CDR (MEMQ '- (EXPLODE X))) (CDR (MEMQ '- E))))
           ((NOT (MEMQ '- E)) (READLIST E))))
```

TMS-NODE returns the TMS node associated with a rule or fact. The error check is useful, in that a frequent mistake is to specify a justification with a constant in the support by forgetting to prefix a variable name with a colon.

```
(DEFUN TMS-NODE (F)
       (IF (SYMBOLP F)
           (LET ((N (COND ((BOUNDP F) (GET-TMS-NODE F))
                          ((TMS-FACTP F) (TMS-FACT-NODE F)))))
                (OR N (ERROR '|BAD ARGUMENT TO TMS-NODE| F 'WRNG-TYPE-ARG)))
           (ERROR '|BAD ARGUMENT TO TMS-NODE| F 'WRNG-TYPE-ARG)))

(DEFUN TMS-NODES (L) (MAPCAR 'TMS-NODE L))
```

The following serve to interface the TMS to AMORD.

```
(DEFUN SUPPORT-STATUS (FACT) (TMS-SUPPORT-STATUS (TMS-NODE FACT)))

(DEFUN ARGUMENT (FACT) (TMS-ANTECEDENT-ARGUMENT (TMS-SUPPORTING-ANTECEDENT (TMS-NODE FACT))))

(DEFUN ANTECEDENT-SET (FACT) (TMS-ANTECEDENT-SET (TMS-NODE FACT)))

(DEFUN SUPPORTING-ANTECEDENT (FACT) (TMS-SUPPORTING-ANTECEDENT (TMS-NODE FACT)))

(DEFUN ANTECEDENTS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ANTECEDENTS (TMS-NODE FACT))))

(DEFUN CONSEQUENCES (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-CONSEQUENCES (TMS-NODE FACT))))

(DEFUN IS-IN (FACT) (TMS-IS-IN (TMS-NODE FACT)))

(DEFUN IS-OUT (FACT) (TMS-IS-OUT (TMS-NODE FACT)))

(DEFUN ARE-IN (FACTS) (TMS-ARE-IN (TMS-NODES FACTS)))

(DEFUN ARE-OUT (FACTS) (TMS-ARE-OUT (TMS-NODES FACTS)))

(DEFUN FOUNDATIONS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ALL-ANTECEDENTS (TMS-NODE FACT))))

(DEFUN REPERCUSSIONS (FACT)
       (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ALL-CONSEQUENCES (TMS-NODE FACT))))

(DEFUN PREMISES (NAME) (MAPCAR 'TMS-EXTERNAL-NAME (TMS-PREMISES (TMS-NODE NAME))))

(DEFUN ASSUMPTIONS (NAME) (MAPCAR 'TMS-EXTERNAL-NAME (TMS-ASSUMPTIONS (TMS-NODE NAME))))

(DEFUN RETRACT (NAME) (TMS-RETRACT (TMS-NODE NAME)))
```

THE UNIFICATION MATCHER

UNIFY takes two expressions and a substitution as input. It returns either a list whose first element is a substitution which yields the most general common unifier of the expressions, relative to the given substitution, if they can be unified, or NIL if they cannot be unified. UNIFY has subroutines for the matching loop, for binding matched variables to values, and for checking for free variable occurrences to avoid erroneous variable capture.

```
(DEFUN UNIFY (A B S)
       ((LAMBDA (S) (AND S (LIST S)))
        (UNIFY-MATCH A B (OR S '(NIL)))))

(DEFUN UNIFY-MATCH (A B S)
       (COND ((EQ A B) S)
             ((ATOM A)
              (AND (NOT (ATOM B)) (VARIABLE B) (UNIFY-VARSET B A S)))
             ((VARIABLE A)
              (UNIFY-VARSET A B S))
             ((ATOM B) NIL)
             ((VARIABLE B) (UNIFY-VARSET B A S))
             (T
              ((LAMBDA (S)
                       (AND S (UNIFY-MATCH (CDR A) (CDR B) S)))
               (UNIFY-MATCH (CAR A) (CAR B) S)))))

(DEFUN UNIFY-VARSET (VAR NEWVAL S)
       (COND ((EQUAL VAR NEWVAL) S)
             (T ((LAMBDA (VCELL)
                         (COND (VCELL (UNIFY-MATCH (CDR VCELL) NEWVAL S))
                               ((UNIFY-FREEFOR VAR NEWVAL S)
                                (CONS (CONS VAR NEWVAL) S))))
                 (ASSOC VAR S)))))

(DECLARE (SPECIAL *CDR-VAR* *E*))

(DEFUN UNIFY-FREEFOR (VAR EXP *E*)
       (LET ((*CDR-VAR* (CDR VAR)))
            (UNIFY-FREEFOR-LOOP EXP)))

(DEFUN UNIFY-FREEFOR-LOOP (E)
       (COND ((ATOM E))
             ((VARIABLE E)
              (AND (NOT (EQ (CDR E) *CDR-VAR*))
                   (UNIFY-FREEFOR-LOOP (CDR (ASSOC E *E*)))))
             (T (AND (UNIFY-FREEFOR-LOOP (CAR E))
                     (UNIFY-FREEFOR-LOOP (CDR E))))))

(DECLARE (UNSPECIAL *CDR-VAR* *E*))
```

INSTANCE takes a pattern and a substitution and returns an expression formed by substituting the substitutions into the pattern and standardizing all variables apart. Boyer-Moore

```
(DECLARE (SPECIAL *SUB* *NEWSUB*))

(DEFUN INSTANCE (EXP *SUB*)
       (LET ((*NEWSUB* NIL)) (INSTANCE-LOOP EXP)))

(DEFUN INSTANCE-LOOP (E)
       (COND ((ATOM E) E)
             ((VARIABLE E)
              (LET ((VCELL (ASSOC E *NEWSUB*)))
                   (COND (VCELL (CDR VCELL))
                         (T (SETQ VCELL (ASSOC E *SUB*))
                            (COND (VCELL (CDAR (SETQ *NEWSUB*
                                                     (CONS
                                                      (CONS E (INSTANCE-LOOP (CDR VCELL)))
                                                      *NEWSUB*))))
                                  (T (CDAR
                                      (SETQ
                                       *NEWSUB*
                                       (CONS
                                        (CONS E (INSTANCE-VGENS (CDR E)))
                                        *NEWSUB*)))))))))
             (T (CONS (INSTANCE-LOOP (CAR E))
                      (INSTANCE-LOOP (CDR E))))))

(DECLARE (UNSPECIAL *SUB* *NEWSUB*))

(DEFUN INSTANCE-VGENS (VNAME)
       (CONS '/: (CONS (CAR VNAME)
                       (SETQ *GENSYM-COUNTER* (+ *GENSYM-COUNTER* 1)))))
```

COMPARE takes two expressions, A and B, as input. If B is a variant of A it returns (VARIANT <substitution>). If A subsumes B it returns (SUBSUMES <substitution>). If B subsumes A it returns (SUBSUMED <substitution>). Otherwise it returns NIL. At any point in the comparison, the state of the comparator may be that either a variant is still possible, or that only either a subsumes or subsumption is possible. These three cases produce the three subroutines of COMPARE.

```
(DECLARE (SPECIAL *TYPE*))

(DEFUN COMPARE (A B)
       (LET ((*TYPE* 'VARIANT))
            (LET ((S (COMPARE-VARIANT-MATCH A B '(NIL))))
                 (AND S (LIST *TYPE* S)))))

(DEFUN COMPARE-VARIANT-MATCH (A B S)
       (COND ((EQ A B) S)
             ((ATOM A) (SETQ *TYPE* 'SUBSUMED) (COMPARE-SUBSUMED-MATCH A B S))
             ((VARIABLE A)
              (COND ((AND (NOT (ATOM B)) (VARIABLE B))
                     (LET ((VCELL (ASSOC A S)))
                          (COND (VCELL
                                 (COND ((EQUAL (CDR VCELL) B) S)
                                       (T (SETQ *TYPE* 'SUBSUMED)
                                          (COMPARE-SUBSUMED-MATCH A B S))))
                                ((RASSOC B S)
                                 (COMPARE-SUBSUMES-MATCH A B S))
                                (T (CONS (CONS A B) S)))))
                    (T (SETQ *TYPE* 'SUBSUMES) (COMPARE-SUBSUMES-MATCH A B S))))
             ((ATOM B) NIL)
             ((VARIABLE B)
              (SETQ *TYPE* 'SUBSUMED)
              (COMPARE-SUBSUMED-MATCH A B S))
             ((SETQ S (COMPARE-VARIANT-MATCH (CAR A) (CAR B) S))
              (COMPARE-VARIANT-MATCH (CDR A) (CDR B) S))))

(DECLARE (UNSPECIAL *TYPE*))

(DEFUN COMPARE-SUBSUMES-MATCH (A B S)
       (COND ((EQ A B) S)
             ((ATOM A) NIL)
             ((VARIABLE A)
              (LET ((VCELL (ASSOC A S)))
                   (COND (VCELL (AND (EQUAL (CDR VCELL) B) S))
                         (T (CONS (CONS A B) S)))))
             ((ATOM B) NIL)
             ((SETQ S (COMPARE-SUBSUMES-MATCH (CAR A) (CAR B) S))
              (COMPARE-SUBSUMES-MATCH (CDR A) (CDR B) S))))
```

```
(DEFUN COMPARE-SUBSUMED-MATCH (A B S)
       (COND ((EQ A B) S)
             ((ATOM B) NIL)
             ((VARIABLE B)
              (LET ((VCELL (RASSOC B S)))
                   (COND (VCELL (AND (EQUAL (CAR VCELL) A) S))
                         (T (CONS (CONS A B) S)))))
             ((ATOM A) NIL)
             ((SETQ S (COMPARE-SUBSUMED-MATCH (CAR A) (CAR B) S))
              (COMPARE-SUBSUMES-MATCH (CDR A) (CDR B) S))))
```

RASSOC is something of an inverse ASSOC, which searches an association list for an association whose CDR matches the supplied key.

```
(DEFUN RASSOC (KEY ALIST)
       (DO ((L ALIST (CDR L))) ((NULL L) NIL)
           (COND ((EQUAL KEY (CDAR L)) (RETURN (CAR L))))))
```

THE DISCRIMINATION NETWORK

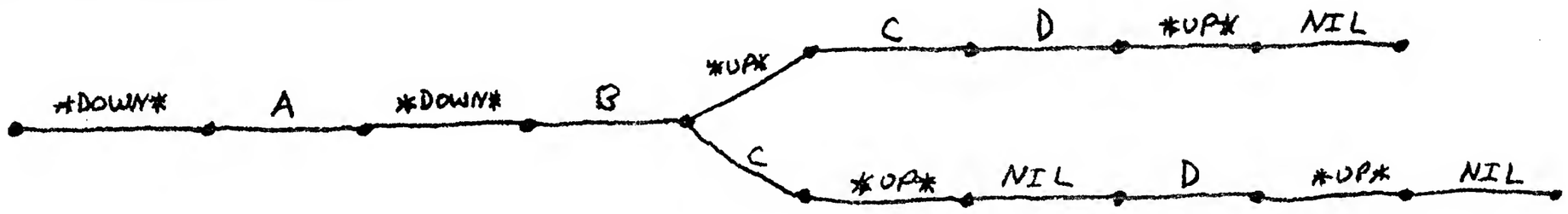
The following functions implement a discrimination net data base. Ignoring the use of the hash table for the moment, let us first understand how a discrimination network is built. Consider the problem of classifying the S-expression (A (B . C) D). Although internally, this expression is a tree, its structure can be expressed as a string of tokens (as for PRINTing it). In this case, the stream of tokens used to discriminate is:

DOWN A *DOWN* B *UP* C D *UP* NIL

A related expression, (A (B C) D), translates into:

DOWN A *DOWN* B C *UP* NIL D *UP* NIL

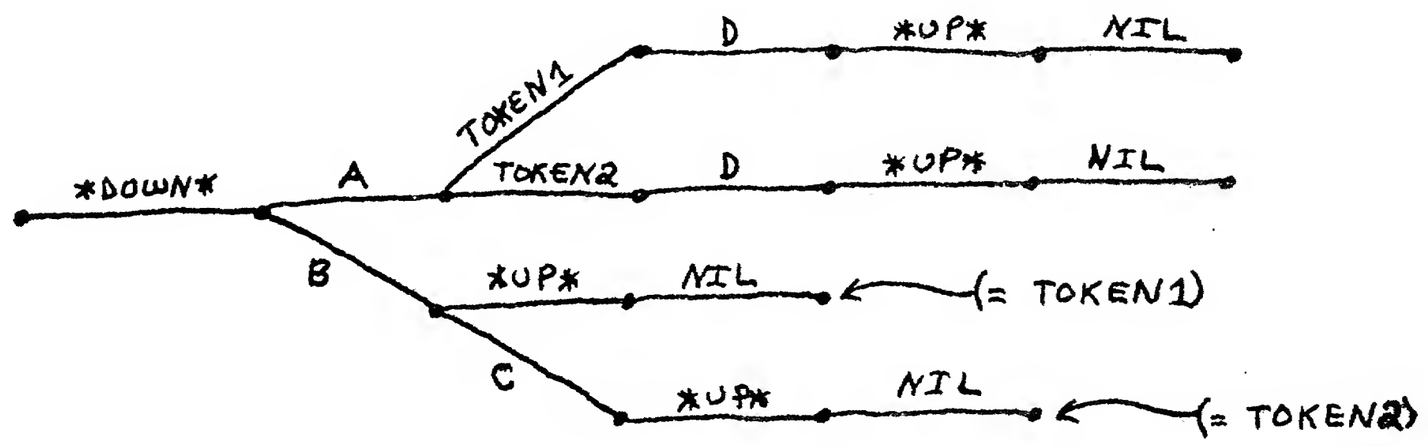
Given these two expressions, we would construct a discrimination net with the following structure:



Given any expression, we extend the discrimination network, if necessary, and return the bucket represented by the appropriate leaf of the discrimination network.

A variable may appear in any position of an expression to be indexed. Each node of the discrimination network contains a special pointer to the subindex for token streams beginning with a variable.

An interesting complexity in this system is that many structures share the same discrimination subnetworks. We assume the user will use lists to represent logic-like terms. These denote the semantic objects being dealt with. It thus makes sense that EQUAL or VARIANT terms be uniquely represented in the network. This is accomplished by discriminating every non-atomic term from the top of the network and then using the resulting bucket as the token for that term in every stream containing that term.



This causes a painful problem: There is now a token for every term, not just every atom. Furthermore, every such token must appear in the top-level node of the network. This makes it unfeasible to use a simple ASSOC of one of these tokens on a part of the node to do a dispatch. Here we introduce a 2-key hash-table to do our associations. Given a token and a discrimination-node, we hash-retrieve an a-list. An element of this a-list beginning with our keys has the required subindex. To introduce further possible bugs, we bubble the association forward in the hash-entry. Donald Duck

There are several global variables in the discrimination net data base. *DN* contains the discrimination net proper, and *HASH-ARRAY* contains the hash table that the discrimination net indexes. *HASH-ARRAY-SIZE* is the size of the hash array, and *DOWN*, *UP*, and *NUMBER* are special tokens used to represent the special types of tokens that construct items entered into the net.

```
(DECLARE (SPECIAL *DN* *DOWN* *UP* *NUMBER* *HASH-ARRAY* *HASH-ARRAY-SIZE*))
```

DBINIT initializes a supplied variable to contain an empty data base.

```
(DEFUN DBINIT ()
       (SETQ *DOWN* (LIST '*DOWN*))
       (SETQ *UP* (LIST '*UP*))
       (SETQ *NUMBER* (LIST '*NUMBER*))
       (SETQ *HASH-ARRAY-SIZE* 1021.)
       (*ARRAY '*HASH-ARRAY* T *HASH-ARRAY-SIZE*)
       (SETQ *DN* (LIST NIL)))
```

STUFF retrieves the list of items from a data base bucket.

```
(DEFUN STUFF (BUCKET) (CDR BUCKET))
```

INSERT-IN-BUCKET does what it says.

```
(DEFUN INSERT-IN-BUCKET (ITEM BUCKET)
       (RPLACD BUCKET (CONS ITEM (CDR BUCKET))))
```

BUCKET returns the bucket of items from a data base corresponding to the supplied expression and substitution, extending the network if necessary to create the bucket for the new expression.

```
(DEFUN BUCKET (EXPRESSION ALIST TYPE)
       (LET ((B (SUB-BUCKET EXPRESSION ALIST *DN*)))
            (OR (HASH-GET TYPE B)
                (LET ((NEWIND (LIST B)))
                     (HASH-PUT NEWIND TYPE)
                     NEWIND)))))
```

SUB-BUCKET does the dirty work for BUCKET by producing the discrimination net token that BUCKET will use to index into the hash-table. The main loop of the program is either to discriminate a list, or to discriminate a thing representing a term -- that is, an atom or a list which is not a sublist of the pattern being indexed. The process of discrimination is termed "walking a path". Variables are not distinguished from each other when discriminating a pattern. If the token being discriminated on is a variable, the unique variable sub-index of the discrimination net node is retrieved and followed. If the token is not a variable, it must be looked up in the table of tokens known at this node. If the token does not exist in the table yet, it is added. The table is maintained in the same hash-table as is used for indexing the buckets. This means that the bubbling of the hash-table entries is constantly rearranging the structure of the discrimination net in accordance with those paths that are followed most frequently.

```
(DECLARE (SPECIAL *ALIST* *INDEX*))

(DEFUN SUB-BUCKET (EXPRESSION *ALIST* *INDEX*)
       (SB-WALK-THING EXPRESSION *INDEX*))

(DEFUN SB-WALK-LIST (FRAGMENT SUBINDEX)
       (COND ((ATOM FRAGMENT)
              (SB-GET-SUBINDEX (IF (NUMBERP FRAGMENT)
                                   *NUMBER*
                                   FRAGMENT)
                               (SB-GET-SUBINDEX *UP* SUBINDEX)))
             ((VARIABLE FRAGMENT)
              (LET ((VCELL (ASSOC FRAGMENT *ALIST*)))
                   (IF VCELL
                       (SB-WALK-LIST (CDR VCELL) SUBINDEX)
                       (SB-GET-VARIABLE-SUBINDEX
                        (SB-GET-SUBINDEX *UP* SUBINDEX)))))
             (T (SB-WALK-LIST (CDR FRAGMENT)
                              (SB-WALK-THING (CAR FRAGMENT) SUBINDEX)))))

(DEFUN SB-WALK-THING (FRAGMENT SUBINDEX)
       (COND ((ATOM FRAGMENT)
              (SB-GET-SUBINDEX (IF (NUMBERP FRAGMENT) *NUMBER* FRAGMENT) SUBINDEX))
             ((VARIABLE FRAGMENT)
              (LET ((VCELL (ASSOC FRAGMENT *ALIST*)))
                   (IF VCELL
                       (SB-WALK-THING (CDR VCELL) SUBINDEX)
                       (SB-GET-VARIABLE-SUBINDEX SUBINDEX))))
             (T (SB-GET-SUBINDEX
                 (SB-WALK-LIST (CDR FRAGMENT)
                               (SB-WALK-THING (CAR FRAGMENT) *INDEX*))
                 (SB-GET-SUBINDEX *DOWN* SUBINDEX)))))

(DECLARE (UNSPECIAL *ALIST* *INDEX*))

(DEFUN SB-GET-SUBINDEX (THING IND)
       (LET ((A (HASH-GET IND THING)))
            (IF A (CDR A)
                (LET ((NEWIND (LIST THING NIL)))
                     (HASH-PUT NEWIND IND)
                     (RPLACD IND (CONS NEWIND (CDR IND)))
                     (CDR NEWIND)))))

(DEFUN SB-GET-VARIABLE-SUBINDEX (IND)
       (OR (CAR IND) (CAR (RPLACA IND (LIST NIL)))))
```

FETCH returns a list of items from a data base which are candidates for unification with the supplied pattern relative to the supplied substitution. In previous versions of this program, FETCH returned a stream which would generate the elements of this list one-by-one. This increased the complexity of the program considerably. The stream version was abandoned due to estimates that the simple list-producing version was more efficient in a system like AMORD, which tries to run every assertion on every rule. FETCH calls on SUB-FETCH to produce a list of indicies into the hash-table corresponding to the list of all tokens in the net which are candidates for matching the supplied pattern. The contents of these buckets are then unioned together and returned.

```
(DEFUN FETCH (PATTERN ALIST TYPE)
       (DO ((L (SUB-FETCH PATTERN ALIST *DN*) (CDR L))
            (ANS
             NIL
             (APPEND (CDR (HASH-GET TYPE (CAR L)))
                     ANS)))
           ((NULL L) ANS)))
```

The complexity of SUB-FETCH derives from the treatment of variables, which can occur in both the fetch patterns *and* in the stored expressions. Variables in the fetch pattern must match only well-formed subexpressions. But expressions are recursively defined sequences of tokens; hence the parenthesis grammar must be counted out. We also allow terminal segments (for example `(A . :X)`) in both patterns and stored expressions. This leads to a case analysis because the initial conditions of the counting argument have to be considered. But all of this analysis serves only to select out those buckets which contain the candidates for the match. Throughout the program, all collected buckets are unioned together (via APPEND, since each item is in a unique bucket), and the resulting list passed back.

Like SUB-BUCKET, SUB-FETCH must walk down the pattern different ways as the item being discriminated is a list or a term-thing. The sub-index retrieval for non-variable tokens is much like that of SUB-BUCKET. The true complexity arises in discriminating variable tokens, since there can be many sub-indicies matching the variable, and the paths corresponding to each of these must be followed. There are two sets of paths to be followed from a variable token, corresponding to the variable matching lists or things.

```
(DECLARE (SPECIAL *ALIST* *INDEX*))

(DEFUN SUB-FETCH (PATTERN *ALIST* *INDEX*)
       (SF-WALK-THING PATTERN (LIST *INDEX*)))

(DEFUN SF-WALK-LIST (FRAGMENT SUBINDICES)
       (COND ((ATOM FRAGMENT)
              (SF-GET-ATOM-SUBINDICES FRAGMENT
                                      (SF-GET-SUBINDICES *UP* SUBINDICES)))
             ((VARIABLE FRAGMENT)
              (LET ((VCELL (ASSOC FRAGMENT *ALIST*)))
                   (IF VCELL (SF-WALK-LIST (CDR VCELL) SUBINDICES)
                       (SF-GET-VARIABLE-LIST SUBINDICES))))
             (T (NCONC (SF-WALK-LIST (CDR FRAGMENT)
                                     (SF-WALK-THING (CAR FRAGMENT) SUBINDICES))
                       (SF-NEXTV (SF-GET-SUBINDICES *UP* SUBINDICES))))))

(DEFUN SF-WALK-THING (FRAGMENT SUBINDICES)
       (COND ((ATOM FRAGMENT)
              (SF-GET-ATOM-SUBINDICES FRAGMENT SUBINDICES))
             ((VARIABLE FRAGMENT)
              (LET ((VCELL (ASSOC FRAGMENT *ALIST*)))
                   (IF VCELL (SF-WALK-THING (CDR VCELL) SUBINDICES)
                       (SF-GET-VARIABLE-THING SUBINDICES))))
             (T (DO ((TOKEN-LIST
                      (SF-WALK-LIST (CDR FRAGMENT)
                                    (SF-WALK-THING (CAR FRAGMENT)
                                                   (LIST *INDEX*)))
                      (CDR TOKEN-LIST))
                     (DOWN-INDICES (SF-GET-SUBINDICES *DOWN* SUBINDICES))
                     (ANS
                      (SF-NEXTV SUBINDICES)
                      (NCONC (SF-GET-SUBINDICES (CAR TOKEN-LIST)
                                                DOWN-INDICES)
                             ANS)))
                    ((NULL TOKEN-LIST) ANS)))))

(DECLARE (UNSPECIAL *ALIST* *INDEX*))

(DECLARE (SPECIAL *THING*))

(DEFUN SF-GET-SUBINDICES (*THING* INDICES)
       (SF-GET-SUBINDICES1 INDICES))

(DEFUN SF-GET-SUBINDICES1 (INDICES)
       (AND INDICES
            (LET ((A (HASH-GET (CAR INDICES) *THING*)))
                 (IF A
                     (CONS (CDR A) (SF-GET-SUBINDICES1 (CDR INDICES)))
                     (SF-GET-SUBINDICES1 (CDR INDICES)))))))
```

```
(DEFUN SF-GET-ATOM-SUBINDICES (TNG INDICES)
       (LET ((*THING* (IF (NUMBERP TNG) *NUMBER* TNG)))
            (SF-GET-ATOM-SUBINDICES1 INDICES)))

(DEFUN SF-GET-ATOM-SUBINDICES1 (INDICES)
       (AND INDICES
            (LET ((A (HASH-GET (CAR INDICES) *THING*)))
                 (COND (A (IF (CAAR INDICES)
                              (CONS (CDR A)
                                    (CONS (CAAR INDICES)
                                          (SF-GET-ATOM-SUBINDICES1 (CDR INDICES))))
                              (CONS (CDR A) (SF-GET-ATOM-SUBINDICES1 (CDR INDICES)))))
                       ((CAAR INDICES)
                        (CONS (CAAR INDICES)
                              (SF-GET-ATOM-SUBINDICES1 (CDR INDICES))))
                       (T (SF-GET-ATOM-SUBINDICES1 (CDR INDICES)))))))

(DECLARE (UNSPECIAL *THING*))

(DEFUN SF-NEXTV (INDICES)
       (COND ((NULL INDICES) NIL)
             ((CAAR INDICES)
              (CONS (CAAR INDICES) (SF-NEXTV (CDR INDICES))))
             (T (SF-NEXTV (CDR INDICES)))))

(DECLARE (SPECIAL *ANS*))

(DEFUN SF-GET-VARIABLE-LIST (INDICES)
       (PROG (*ANS*)
             (MAPC 'SF-GVL INDICES)
             (RETURN *ANS*)))

(DEFUN SF-GVL (I)
       (MAPC '(LAMBDA (ASUB)
                      (COND ((EQ (CAR ASUB) *UP*)
                             (MAPC '(LAMBDA (AS) (SETQ *ANS* (CONS (CDR AS) *ANS*)))
                                   (CDDR ASUB))
                             (AND (CADR ASUB) (SETQ *ANS* (CONS (CADR ASUB) *ANS*))))
                            ((EQ (CAR ASUB) *DOWN*)
                             (MAPC '(LAMBDA (AS) (SF-GVL (CDR AS))) (CDDR ASUB))
                             (AND (CADR ASUB) (SF-GVL (CADR ASUB))))
                            (T (SF-GVL (CDR ASUB)))))
             (CDR I))
       (AND (CAR I) (SF-GVL (CAR I))))

(DECLARE (UNSPECIAL *ANS*))
```

```
(DEFUN SF-GET-VARIABLE-THING (INDICES)
       (PROG (ANS)
             (MAPC '(LAMBDA (I)
                            (MAPC '(LAMBDA (ASUB)
                                           (COND ((EQ (CAR ASUB) *UP*) NIL)
                                                 ((EQ (CAR ASUB) *DOWN*)
                                                  (MAPC '(LAMBDA (AS)
                                                                 (SETQ ANS (CONS (CDR AS)
                                                                                 ANS)))
                                                        (CDDR ASUB))
                                                  (IF (CADR ASUB)
                                                      (SETQ ANS (CONS (CADR ASUB) ANS))))
                                                 (T (SETQ ANS (CONS (CDR ASUB) ANS)))))
                                  (CDR I))
                            (IF (CAR I) (SETQ ANS (CONS (CAR I) ANS))))
                   INDICES)
             (RETURN ANS)))
```

The following functions implement the hash table for associations used in making the token dispatch step of the discrimination more efficient.

```
(DECLARE (FIXNUM *HASH-ARRAY-SIZE* (HASH-NUMBER NOTYPE NOTYPE) NUM)
         (ARRAY* (NOTYPE (*HASH-ARRAY* ?))))
```

HASH-GET retrieves a specified thing from the hash table of the supplied data base.

```
(DEFUN HASH-GET (INDEX THING)
       (CDR (2-BSSQ INDEX THING
                    (*HASH-ARRAY* (HASH-NUMBER INDEX THING)))))
```

HASH-PUT inserts a new thing into the hash table of the given data base.

```
(DEFUN HASH-PUT (NEWINDEX INDEX)
       ((LAMBDA (NUM)
                (STORE (*HASH-ARRAY* NUM)
                       (CONS (CONS INDEX NEWINDEX)
                             (*HASH-ARRAY* NUM))))
        (HASH-NUMBER INDEX (CAR NEWINDEX))))
```

This is the ubiquitous number computer.

```
(DEFUN HASH-NUMBER (KEY1 KEY2)
       (\ (BOOLE 6 (MAKNUM KEY1) (MAKNUM KEY2))          ;XOR
          *HASH-ARRAY-SIZE*))
```

2-BSSQ searches an association list for an association of the pairing of the supplied two keys, and for efficiency [Rivest 1976], bubbles the association one step towards the front of the association list.

```
(DEFUN 2-BSSQ (K1 K2 L)
       (PROG (L1 L2)
             (COND ((NULL L) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L)) (EQ K2 (CADAR L)))
                    (RETURN (CAR L))))
             (SETQ L2 L)
          LP (SETQ L1 (CDR L2))
             (COND ((NULL L1) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L1)) (EQ K2 (CADAR L1)))
                    (RPLACA L2
                            (PROG2 NIL (CAR L1)
                                   (RPLACA L1 (CAR L2))))
                    (RETURN (CAR L2))))
             (SETQ L2 (CDR L1))
             (COND ((NULL L2) (RETURN NIL))
                   ((AND (EQ K1 (CAAR L2)) (EQ K2 (CADAR L2)))
                    (RPLACA L1
                            (PROG2 NIL (CAR L2)
                                   (RPLACA L2 (CAR L1))))
                    (RETURN (CAR L1))))
             (GO LP)))
```

This concludes the listing of the interpreter.

Notes

AMORD

A Miracle of Rare Device, a name taken (by Doyle) from S. T. Coleridge's poem Kubla Khan.

Donald Duck

If you think the structure of our discrimination network is devious, you should see the previous version, which generates candidates incrementally. But even that program doesn't hold a candle to Drew McDermott's Donald Duck discrimination network!

Explicit Control

A more detailed discussion of the technique of explicit control encouraged by AMORD can be found in [de Kleer, Doyle, Steele and Sussman 1977].

Godel

Self-referential facts cannot be recognized, as the order in which rule environments are constructed precludes rules with patterns like (:F (CRETIN :F)).

Boyer-Moore

Doyle and Sussman experimented with the use of the Boyer-Moore structure sharing implementation of assertions. In benchmark tests it was found that (in the current implementation) the average rule consumed some 20 words less than the average assertion. Since the only real difference is that rules share structure, while each assertion has its own instance of its pattern, this led to hopes of space saving by moving to a more efficient representation. Unfortunately, calculations showed that this more complicated scheme would not result in very significant space savings. In addition, its implementation seems to entail a very significant amount of computation in a system like AMORD, in which new assertions must be checked against the data base for subsumptions. While the routines for unification and instancing are simple to write and execute efficiently, the comparison routines seem to be much more complicated and very much less efficient. Our experience with the Boyer-Moore representation should be compared with that of McDermott [1977].

MacLISP

MacLISP [Moon 1974] is a powerful dialect of LISP developed by the MIT Artificial Intelligence Laboratory.

TMS

The Truth Maintenance System is a program developed by Doyle [1978a,b]. Section 3 summarizes its function and use.

References

[de Kleer, Doyle, Steele and Sussman 1977]
Johan de Kleer, Jon Doyle, Guy L. Steele Jr., and Gerald Jay Sussman, "Explicit Control of Reasoning," MIT AI Lab, Memo 427, June 1977.

[Doyle 1978a]
Jon Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab TR-419, January 1978.

[Doyle 1978b]
Jon Doyle, "A Glimpse of Truth Maintenance," MIT AI Lab Memo 461, February 1978.

[McDermott 1977]
Drew Vincent McDermott, "Flexibility and Efficiency in a Computer Program for Designing Circuits," MIT AI Lab TR-402, June 1977.

[Moon 1974]
David A. Moon, "MacLISP Reference Manual," MIT Project Mac, Revision 0, April 1974.

[Rivest 1976]
Ronald Rivest, "On Self-Organizing Sequential Search Heuristics," *CACM 19, #2*, (February 1976), pp. 63-67.

[Sussman and Stallman 1975]
Gerald Jay Sussman and Richard Matthew Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, No. 11, November 1975, pp. 857-865.